Type Theory
(the very basics)
What is Type Theory?
What is Type Theory?

Type Theory is a logic.
What is Type Theory?

[1971]
Per Martin-Löf

Type Theory is a logic.
What is Type Theory?

[1971]
Per Martin-Löf

Type Theory is a logic.

- but, being based on $\lambda$-calculus, is bares close similarity to programming languages.
Type Theory is a Logic

• As a logic, it is interested in making judgements of the form:
  
  ▶ A is a set
  ▶ \( A_1 \) and \( A_2 \) are equal sets
  ▶ \( a \) is in an element of set \( A \)
  ▶ \( a_1 \) and \( a_2 \) are equal elements in \( A \)
Judgement Forms

• To know that $A$ is a set is to know how to form the canonical elements in the set and under what conditions two canonical elements are equal.
Judgement Forms

• To know that $A$ is a set is to know how to form the canonical elements in the set and under what conditions two canonical elements are equal.

Set of natural numbers

\[
\begin{aligned}
data \text{ nat} & \triangleq \\
0 & : \text{ nat} \\
S : \text{ nat} & \rightarrow \text{ nat}
\end{aligned}
\]

\[
\begin{aligned}
0 \\
1 & \triangleq S \ 0 \\
2 & \triangleq S \ (S \ 0) \\
3 & \triangleq S \ (S \ (S \ 0)) \\
& \vdots
\end{aligned}
\]
Judgement Forms

• To know that $A$ is a set is to know how to form the canonical elements in the set and under what conditions two canonical elements are equal.

Set of natural numbers

\begin{align*}
\textbf{data} \ & \text{nat} \triangleq \\
0 : \text{nat} \\
S : \text{nat} \rightarrow \text{nat}
\end{align*}

\begin{align*}
0 \\
1 & \triangleq S 0 \\
2 & \triangleq S (S 0) \\
3 & \triangleq S (S (S 0)) \\
\vdots
\end{align*}

(Polymorphic) set of lists

\begin{align*}
\textbf{data} \ & \text{List E} \triangleq \\
\text{Nil : List E} \\
\text{Cons : E} \rightarrow \text{List E} \rightarrow \text{List E}
\end{align*}

\begin{align*}
[] & \triangleq \text{Nil} \\
[a_1] & \triangleq \text{Cons a}_1 \text{ Nil} \\
[a_1, a_2] & \triangleq \text{Cons a}_1 (\text{Cons a}_2 \text{ Nil}) \\
[a_1, a_2, a_3] & \triangleq \text{Cons a}_1 (\text{Cons a}_2 (\text{Cons a}_3 \text{ Nil})) \\
\vdots
\end{align*}
Judgement Forms

• If A is a set then to know that \( a \in A \) is to know that a, when evaluated, yields a canonical element in A as value.
Judgement Forms

• If A is a set then to know that $a \in A$ is to know that $a$, when evaluated, yields a canonical element in $A$ as value.

\[
\text{(fix } \lambda h \ n. \ \text{match } n \ \text{with}} \\
\quad \quad \quad \quad 0 \Rightarrow \text{Nil} \\
\quad \quad \quad S \ k \Rightarrow \text{Cons } n \ (h \ k) \ \text{)} \ 8 \ : \ \text{List } \text{nat}
\]
Judgement Forms

• If A is a set then to know that \( a \in A \) is to know that a, when evaluated, yields a canonical element in A as value.

\[
\text{(fix } \lambda h \ n. \ \text{match } n \ \text{with} \\
0 \Rightarrow \text{Nil} \\
S \ k \Rightarrow \text{Cons } n \ (h \ k) \ ) \ 8 \ : \ \text{List nat}
\]

› We know this from our typing rules and from properties of the type system (progress & preservation)
Judgement Forms

• Quite unsurprisingly then,

  ▶ *Sets* are characterized by *types*,

  ▶ *Membership* is characterized by *typing rules*. 
Propositions in Type Theory

- Propositions are **types**.
  - A member of the type represents a **proof** of the proposition.
    (sometimes called a **witness**.)

\[
\text{t : } 3 > 2 \quad \text{t is a proof that } 3 > 2 \text{ is valid.}
\]
Propositions in Type Theory

• Propositions are **types**.
  - A member of the type represents a **proof** of the proposition.
    (sometimes called a **witness**.)

\[
t : 3 > 2
\]

\( t \) is a **proof** that \( 3 > 2 \) is valid.
Propositions in Type Theory

- Propositions are **types**.
  - A member of the type represents a **proof** of the proposition.
    (Sometimes called a **witness**.)

\[
\begin{align*}
  t & : 3 > 2 \\
  3 > 2 & : \mathbb{P}
\end{align*}
\]

- Propositions have a special type called **Prop**, or \( \mathbb{P} \).
Propositional Logic

Logical True

Logical False

traditional logic

type theory
Propositional Logic

Logical $True$
\[ \top \]

Logical $False$
\[ \bot \]

**data** $True \triangleq$

$I : True$

unit type

diagram:

- [Logical $True$]
  - $\top$

- [Logical $False$]
  - $\bot$

*traditional logic*  *type theory*
Propositional Logic

Logical \( \text{True} \)
\[ \top \]

Logical \( \text{False} \)
\[ \bot \]

\textbf{traditional logic}

\textbf{type theory}

\textbf{data } \text{True} \triangleq
\[ I : \text{True} \]
unit type

\textbf{data } \text{False} \triangleq
\[ \leftrightarrow \text{ nothing} \]
bottom type

type with \textit{no constructors}
Propositional Logic

Conjunction
\[ A \land B \]

Disjunction
\[ A \lor B \]
Propositional Logic

Conjunction
A \land B

data Pair A B \triangleq pair : A \rightarrow B \rightarrow Pair A B
product types (A * B)

Disjunction
A \lor B

traditional logic
type theory
Propositional Logic

Conjunction
\( A \land B \)

Disjunction
\( A \lor B \)

data Pair A B \(\triangleq\)
\[
\begin{align*}
\text{pair} & : A \rightarrow B \rightarrow \text{Pair} A B \\
\text{product types} & (A \ast B)
\end{align*}
\]
data Either A B \(\triangleq\)
\[
\begin{align*}
\text{inl} & : A \rightarrow \text{Either} A B \\
\text{inr} & : B \rightarrow \text{Either} A B \\
\text{sum types} & (A + B)
\end{align*}
\]
Propositional Logic

Implication

\[ P \rightarrow Q \]
Propositional Logic

Implication

\[ P \rightarrow Q \]

function types!

\[ P \rightarrow Q \]

traditional logic

type theory
Propositional Logic

Implication

\[ P \rightarrow Q \]

Negation

\[ \neg P \]

function types!

traditional logic

type theory
Propositional Logic

Implication

\[ P \rightarrow Q \]

Negation

\[ \neg P \]

traditional logic

\[ P \rightarrow \text{False} \]

function to vacuity

\[ P \rightarrow Q \]

function types!

\[ P \rightarrow \text{False} \]

type theory
First-Order Logic

Function Symbols

\[ f : S_1 \times S_2 \rightarrow S_3 \]

Predicates

\[ R : S_1 \times S_2 \]

traditional logic type theory
First-Order Logic

Function Symbols
\[ f : S_1 \times S_2 \to S_3 \]

Predicates
\[ R : S_1 \times S_2 \]

well... functions.

traditional logic

type theory
First-Order Logic

Function Symbols
\[ f : S_1 \times S_2 \rightarrow S_3 \]

Well-function.

Predicates
\[ R : S_1 \times S_2 \]

Functions to Prop

traditional logic  type theory
First-Order Logic

Equality
\[ t_1 = t_2 \]

**data** \[ \text{eq} \ A : A \to A \to \mathbb{P} \triangleq \]

\[ \text{eq}_\text{refl} : \forall a : A. \text{eq} \ a \ a \]

definitional equality

*traditional logic*  

*type theory*
First-Order Logic

Equality
\[ t_1 = t_2 \]

\[ \text{data } \text{eq } A : A \to A \to \mathbb{P} \triangleq \text{eq_refl } : \forall a : A. \text{eq } a \ a \]

\[ \text{eq_refl } 8 : 8 = 8 \]

definitional equality

\[ \text{eq } \text{nat } 8 \ 8 \equiv 8 = 8 : \mathbb{P} \]

\[ \text{eq}_\text{refl } 8 \text{ is a proof that } 8 = 8 \text{ is valid.} \]

traditional logic

type theory
Quantifiers...

Universal Quantifier

\[ \forall x : S. P(x) \]

Existential Quantifier

\[ \exists x : S. P(x) \]

traditional logic

\textit{type theory}
Quantifiers...

Universal Quantifier
\[ \forall x : S. P(x) \]

Existential Quantifier
\[ \exists x : S. P(x) \]

dependent function types
(they even have the same syntax...)

\[ \forall x : S. P x \]

traditional logic  

type theory
Quantifiers...

Universal Quantifier
\[ \forall x : S. P(x) \]

Existential Quantifier
\[ \exists x : S. P(x) \]

dependent function types
(they even have the same syntax...)

traditional logic

type theory
Quantifiers...

Universal Quantifier
\[ \forall x : S. \, P(x) \]

Existential Quantifier
\[ \exists x : S. \, P(x) \]

dependent function types
(they even have the same syntax...)

\[ \forall x : S. \, P(x) \]

\[ \text{data} \, \text{Sig} \, S \, P \triangleq \]
\[ \text{sig} : \forall x : S. \, P(x) \rightarrow \text{Sig} \, A \, B \]

dependent pair types

\[ \text{traditional logic} \]

\[ \text{type theory} \]
Quantifiers...

Universal Quantifier
\[ \forall x : S. P(x) \]

Existential Quantifier
\[ \exists x : S. P(x) \]

dependent function types
(they even have the same syntax...)

dependent pair types

data Sig S P ≜
\[ \text{sig} : \forall x : S. P(x) \rightarrow \text{Sig A B} \]

traditional logic

\[ : P \]

type theory
Quantifiers...

Universal Quantifier
\[ \forall x : S. \, P(x) \]

Existential Quantifier
\[ \exists x : S. \, P(x) \]

(\(\Pi\) types)
\[ \forall x : S. \, P(x) \]
dependent function types
(they even have the same syntax...)

(data)
\[ \text{Sig } S \, P \]
\[ \text{sig} : \forall x : S. \, P(x) \rightarrow \text{Sig } A \, B \]

(\(\Sigma\) types)
dependent pair types

traditional logic

\[ \text{type theory} \]
Quantifiers...

\[ P \rightarrow Q \]
function types

\[ \forall x : S. P x \]
dependent function types

\[ \wedge \]
data Pair A B \triangleq 
pair : A \rightarrow B \rightarrow \text{Pair A B} 
product types (A * B)

\[ \exists \]
data Sig S P \triangleq 
sig : \forall x : S. P x \rightarrow \text{Sig A B} 
dependent pair types

*type theory*
Quantifiers: an aside

• In Type Theory:

\( \forall \) is a dependent generalization of \( \rightarrow \)

\( \exists \) is a dependent generalization of \( \wedge \)
High-Order Quantifiers..!

Universal Quantifier

$$\forall x : S. P(x)$$

(Π types)

$$\forall x : S. P x$$

dependent function types

$$\forall A B (f : A \rightarrow B) x y. \ x = y \rightarrow f x = f y \ : \ P$$

traditional logic  

| type theory |
Beyond First-Order Logic

Induction?
For \( n = 0 \): ...
Assume for \( n = k \),
prove for \( n = k+1 \):
...

traditional logic

type theory
Induction?
For \( n = 0 \): ...
Assume for \( n = k \),
prove for \( n = k+1 \):
...

\[
\text{fix } \lambda \text{IH } n. \ \text{match } n \text{ with}
\]
\[
0 \Rightarrow \quad \ldots \quad : P \ 0
\]
\[
S \ k \Rightarrow \quad \ldots (\text{IH } k) \ldots \quad : P \ (S \ k)
\]

Recursion!

\[ \text{traditional logic} \] \hspace{1.5cm} \[ \text{type theory} \]
Beyond First-Order Logic

Induction?

For \( n = 0 \): ...
Assume for \( n = k \),
prove for \( n = k+1 \):
...

\[
\text{fix } \lambda \text{IH } n. \text{ match } n \text{ with }
\begin{align*}
0 & \Rightarrow \quad \ldots \quad : P \ 0 \\
S \ k & \Rightarrow \quad \ldots \ (\text{IH } k) \ldots \quad : P \ (S \ k)
\end{align*}
\]
Recursion!

traditional logic  

\textbf{type theory}
Beyond First-Order Logic

Induction?
For \( n = 0 \): ...
Assume for \( n = k \),
prove for \( n = k+1 \):
...

\[
\text{fix } \lambda \text{IH } n. \quad \text{match } n \text{ with } \\
0 \Rightarrow \quad \ldots \\
S k \Rightarrow \quad \ldots (\text{IH } k) \ldots \\
\text{Recursion!}
\]

\[
\forall n : \mathbb{N}. \quad P \ n \\
\]

traditional logic

type theory
Beyond First-Order Logic

Induction?
For \( n = 0 \): ...
Assume for \( n = k \),
prove for \( n = k+1 \):
...

Induction schema

\[
\begin{align*}
P(0) & \quad P(k) \rightarrow P(k+1) \\
\hline
P(n) & \\
\end{align*}
\]

Recursion!

\[
\text{fix } \lambda \text{IH } n. \quad \text{match } n \text{ with}
\begin{align*}
0 & \Rightarrow \ldots \quad : P 0 \\
S \ k & \Rightarrow \ldots (\text{IH } k) \ldots \quad : P (S \ k)
\end{align*}
\]

Induction principle

\[
\text{nat_ind} : \forall P : \mathbb{N} \rightarrow P.
\begin{align*}
P \ 0 & \rightarrow (\forall k : \mathbb{N} . \ P \ k \rightarrow P (S \ k)) \\
& \rightarrow \forall n : \mathbb{N}, \ P \ n
\end{align*}
\]

traditional logic

type theory
Curry-Howard Correspondence
Curry-Howard Correspondence

Haskell B. Curry
[1969]
Curry-Howard Correspondence

Haskell B. Curry [1969]

William A. Howard [1934, 1958]
Curry-Howard Correspondence

Haskell B. Curry [1969]

William A. Howard [1934, 1958]
## Curry-Howard Correspondence

<table>
<thead>
<tr>
<th>Logic</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositions / statements</td>
<td>Types</td>
</tr>
<tr>
<td>Proofs</td>
<td>Programs</td>
</tr>
<tr>
<td>True statement</td>
<td>Unit type</td>
</tr>
<tr>
<td>False statement</td>
<td>Bottom (empty) type</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Product type (pair)</td>
</tr>
<tr>
<td>Disjunction</td>
<td>Sum type (either)</td>
</tr>
<tr>
<td>Implication</td>
<td>Function type</td>
</tr>
<tr>
<td>Universal quantification</td>
<td>Dependent function type</td>
</tr>
<tr>
<td>Existential quantification</td>
<td>Dependent pair type</td>
</tr>
<tr>
<td>Induction</td>
<td>Recursion</td>
</tr>
</tbody>
</table>
3-Step Plan for Proving Propositions

∀n : N. 0 ≤ n : P
3-Step Plan for Proving Propositions

- **Step 1.** Write program

```latex
\textbf{fix} \quad \forall \text{IH } n. \quad \text{\textbf{match} } n \text{ \textbf{with}} \quad \\
0 \quad \Rightarrow \quad \text{le}_n 0 \quad \\
\text{S } k \quad \Rightarrow \quad \text{le}_S 0 \ k \ (\text{IH } k) \quad \forall n : \mathbb{N}. \ 0 \leq n \ : \ P
```

- **Step 1.** Write program
3-Step Plan for Proving Propositions

- **Step 1.** Write program
  
  ```
  fix \lambda IH n. 
  match n with
  0 \Rightarrow \text{le}_n 0 
  S k \Rightarrow \text{le}_S 0 k \text{ (IH k)}
  ```

- **Step 2.** Type-check

  - : \( 0 \leq 0 \)
  - : \( 0 \leq S k \)
  - : \( \forall n : \mathbb{N}. \ 0 \leq n : \mathbb{P} \)
3-Step Plan for Proving Propositions

\[
\begin{align*}
\text{Step 1. } & \text{Write program} \\
\text{Step 2. } & \text{Type-check} \\
\text{Step 3. } & \text{Profit!}
\end{align*}
\]

\[
\text{fix } \lambda \text{IH } n. \\
\text{match } n \text{ with} \\
\quad 0 \Rightarrow \text{le}_n 0 \\
\quad S \ k \Rightarrow \text{le}_S 0 \ k \ (\text{IH } k)
\]

\[
: 0 \leq 0 \\
: 0 \leq S \ k \\
: \forall n : \mathbb{N}. 0 \leq n : \mathbb{P}
\]
3-Step Plan for Proving Propositions

**Step 1.** Write program

\[
\text{fix } \lambda (\text{IH} : \forall n : \mathbb{N}. 0 \leq n) \ n. \\
\text{IH} \ n
\]

**Step 2.** Type-check

\[
\forall n : \mathbb{N}. 0 \leq n : \mathbb{P}
\]

**Step 3.** Profit?

Errhm….
3-Step Plan for Proving Propositions

∀n : N. 0 ≤ n : \( P \)
3-Step Plan for Proving Propositions

- **Step 1.** Write program

\[
\text{fix } \lambda \text{IH } n. \\
\text{match } n \text{ with} \\
\quad 0 \Rightarrow \text{le}_n 0 \\
\quad S \ k \Rightarrow \text{le}_S 0 \ k \ (\text{IH } k)
\]

\[\forall n : \mathbb{N}. \ 0 \leq n \quad : \quad P\]
3-Step Plan for Proving Propositions

- **Step 1.** Write program
  
  ```
  fix \lambda IH n. 
  match n with 
  0 \Rightarrow le_n 0
  S k \Rightarrow le_S 0 k (IH k)
  ```

- **Step 2.** Type-check
  
  \[ \forall n : \mathbb{N}. 0 \leq n : \mathbb{P} \]

  - \( : 0 \leq 0 \)
  - \( : 0 \leq S k \)
3-Step Plan for Proving Propositions

- **Step 1.** Write program
  
  ```
  fix \lambda IH n. 
  match n with 
  0 => le_n 0 
  S k => le_S 0 k (IH k) 
  ```

- **Step 2.** Type-check
  
  - `0 ≤ 0`
  - `0 ≤ S k`

- **Step 3.** Termination check
  
  `∀ n : \mathbb{N}. 0 ≤ n : \mathbb{P}`
Constructivism

• Type Theory is a form of constructive mathematics.
  
  ‣ To prove $A \lor B$:
    one has to show how to determine whether $A$ is true or $B$ is true.
  
  ‣ To prove $\exists x. P$:
    one has to show how to find a value of $x$ that satisfies $P$. 
Constructivism

• Type Theory is a form of constructive mathematics.
  ▸ To prove $A \lor B$: one has to show how to determine whether $A$ is true or $B$ is true.
  ▸ To prove $\exists x. P$: one has to show how to find a value of $x$ that satisfies $P$. 
Constructivism

- Type Theory is a form of constructive mathematics.
  - To prove $A \lor B$: one has to show how to determine whether $A$ is true or $B$ is true.
  - To prove $\exists x. P$: one has to show how to find a value of $x$ that satisfies $P$.

Compute (algorithm!)
Implementation

Coq

Thierry Coquand

CoC

[1988]

Calculus of Constructions

later CIC

Calculus of Inductive Constructions
Implementation

Coq

nat list A le x y A \ B

Type

Set Prop
Lab 4

• Prove some properties of natural numbers
  ‣ Follow the lemmas in gcd.v and fill in the proofs

• Prove (partial) correctness of a “softened” version of Euclid’s GCD algorithm.
  ‣ (gcd_correct) If gcd terminates, then the return value is a GCD according to the predicate is_gcd
    ○ Use auxiliary lemma gcd_step_aux, then apply it in the step_a and step_b branches of the induction on gcd.